View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The Python Papers Anthology

COPADS, II: Chi-Square test, F-Test and t-Test Routines from *Gopal Kanji's* 100 Statistical Tests

Zhu En Chay[1] and Maurice HT Ling [1, 2]
[1]*School of Chemical and Life Sciences, Singapore Polytechnic, Singapore*
[2]*Department of Zoology, The University of Melbourne, Australia*
mauriceling@acm.org

Abstract

This manuscript presents the implementation and testing of 8 Chi-Square test, 3 F-test and 6 T-test routines from *Gopal Kanji*'s book entitled "100 Statistical Tests".

1. Introduction

Kanji (2006)'s 100 Statistical Tests is a collection of commonly used statistical tests, ranging from single-sample to multi-samples parametric and non-parametric tests, with sample calculation provided for each test. Ten of the hundred tests had been implemented by Ling (2009). This manuscript is a continuation of Ling (2009a)'s Ten Z-test Routines from Gopal Kanji's 100 Statistical Tests.

This manuscript presents the implementation (file name: AllTests.py) and testing (file name: AllTestfile.py) of 8 Chi-Square tests, 3 F-tests and 6 t-tests, where the details and limitations of each test are given as docstrings in the codes below. A statistical test harness is implemented (AllTestfile.test function) to standardize implementation of each hypothesis testing routines. This harness can be used for both 1-tailed or 2-tailed hypothesis tests as it reports both, the upper and lower critical value, with the given confidence level and state whether the calculated test statistic is more than the upper critical value or lower than the lower critical value or neither (Ling, 2009b).

Chi-Square tests are statistical tests that are commonly used to determine whether there are significant differences between expected frequencies and the observed frequencies in one or more categories (Sirkin, 2006). The test statistic follows a Chi-square distribution with (k – c) degrees of freedom where k is the number of non-empty cells and c = the number of estimated parameters (Sirkin, 2006).

F-tests are statistical tests using F distribution under a null hypothesis through the comparison of the ratio of two variances (Downward, 2006). It is used when comparing statistical models that have been fit to a data set to determine models that best fit the population from the data being sampled (Downward, 2006). The F distribution is a joint

distribution of two independent variables, with each having a Chi-Square distribution (Ling, 2009b).

T-tests are statistical hypothesis tests that follow a Student's t distribution if the null hypothesis is true (McDonald, 2008). The Student's t-distribution is a continuous probability distribution that is used to estimate the mean of a normally distributed population when the sample size is large (McDonald, 2008). By computing the t-statistic, a p-value can be determined, displaying the extent of difference of the means between two populations where the smaller the p-value, the more significant is the difference between the distributions of the two populations (McDonald, 2008).

These codes are collected under "Collection of Python Algorithms and Data Structures" (COPADS, http://copads.sf.net) and licensed under Python Software Foundation License version 2.

2. Code Files

File: AllTests.py

```
"""
This file contains the implementation of 8 Chi-Square test, 3 F-test
and 6 T-test routines from Gopal Kanji's book and published
under The Python Papers Source Codes.

The implemented statistical tests are:
1. Test 7: t-test for a population mean (population variance unknown)
2. Test 8: t-test for two population means (population variance unknown
but equal)
3. Test 9: t-test for two population means (population variance unknown
and unequal)
4. Test 10: t-test for two population means (method of paired comparisons)
5. Test 11: t-test of a regression coefficient
6. Test 12: t-test of a correlation coefficient
7. Test 15: Chi-square test for a population variance
8. Test 16: F-test for two population variances (variance ratio test)
9. Test 17: F-test for two population variances (with correlated
observations)
10. Test 24: Chi-square test for an assumed population variance
11. Test 25: F-test for two counts (Poisson distribution)
12. Test 37: Chi-square test for goodness of fit
13. Test 38: The x2-test for compatibility of K counts
14. Test 40: Chi-square test for consistency in 2x2 table
15. Test 41: The x2-test for consistency in a K x 2 table
16. Test 43: The x2-test for consistency in a 2 x K table
17. Test 50: The median test of two populations

These codes are licensed under Python Software Foundation License version
2.
```

```
"""
from copadsexceptions import DistributionParameterError
from copadsexceptions import DistributionFunctionError
from copadsexceptions import NormalDistributionTypeError
import nrpy
from math import sqrt, log, e, exp
SQRT2 = 1.41421356237309504880168872420969807856967187537694
PI2 = 6.28318530717958647692528676655900576839433879875021


class Distribution:
    """
    Abstract class for all statistical distributions.
    Due to the large variations of parameters for each distribution, it is
    unlikely to be able to standardize a parameter list for each method
    that is meaningful for all distributions. Instead, the parameters to
    construct each distribution is to be given as keyword arguments.

    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4
    """

    def __init__(self, **parameters):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.
        """
        raise NotImplementedError

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity or 0
        to a give x-value on the x-axis where y-axis is the probability.
        CDF is also known as density function.
        """
        raise NotImplementedError

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        raise NotImplementedError

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        raise NotImplementedError

    def mean(self):
        """
        Gives the arithmetic mean of the sample.
```

```
        """
        raise NotImplementedError

    def mode(self):
        """
        Gives the mode of the sample, if closed-form is available.
        """
        raise NotImplementedError


    def kurtosis(self):
        """
        Gives the kurtosis of the sample.
        """
        raise NotImplementedError

    def skew(self):
        """
        Gives the skew of the sample.
        """
        raise NotImplementedError

    def variance(self):
        """
        Gives the variance of the sample.
        """
        raise NotImplementedError

class FDistribution(Distribution):
    """
    Class for F Distribution.

    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4
    """

    def __init__(self, df1=1, df2=1):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param df1: degrees of freedom for numerator
        @param df2: degrees of freedom for denorminator
        """
        self.df1 = float(df1)
        self.df2 = float(df2)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity or 0
        to a give x-value on the x-axis where y-axis is the probability.
        """
        sub_x = (self.df1 * x) / (self.df1 * x + self.df2)
        return nrpy.betai(self.df1 / 2.0, self.df2 / 2.0, sub_x)
```

```
    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability
        for particular value of x, or the area under probability
        distribution from x-h to x+h for continuous distribution.
        """
        x = float(x)
        n1 = ((x * self.df1) ** self.df1) * (self.df2 ** self.df2)
        n2 = (x * self.df1 + self.df2) ** (self.df1 + self.df2)
        d = x * nrpy.beta(self.df1 / 2.0, self.df2 / 2.0)
        return math.sqrt(n1 / n2) / d

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        And the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob:
            return (start, cprob)
        while probability > cprob:
            start = start + step
            cprob = self.CDF(start)
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return float(self.df2 / (self.df2 - 2))

class GammaDistribution(Distribution):
    """
    Class for Gamma Distribution.

    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4
    """

    def __init__(self, location, scale, shape):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param location:
        @param scale:
        @param shape:"""
        self.location = float(location)
        self.scale = float(scale)
        self.shape = float(shape)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity or 0
        to a give x-value on the x-axis where y-axis is the probability.
        """
        return nrpy.gammp(self.shape, (x - self.location) / self.scale)
```

```
    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        And the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob:
            return (start, cprob)
        while probability > cprob:
            start = start + step
            cprob = self.CDF(start)
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return self.location + (self.scale * self.shape)

    def mode(self):
        """Gives the mode of the sample."""
        return self.location + (self.scale * (self.shape - 1))

    def kurtosis(self):
        """Gives the kurtosis of the sample."""
        return 6 / self.shape

    def skew(self):
        """Gives the skew of the sample."""
        return 2 / math.sqrt(self.shape)

    def variance(self):
        """Gives the variance of the sample."""
        return self.scale * self.scale * self.shape

    def qmean(self):
        """Gives the quantile of the arithmetic mean of the sample."""
        return nrpy.gammp(self.shape, self.shape)

    def qmode(self):
        """Gives the quantile of the mode of the sample."""
        return nrpy.gammp(self.shape, self.shape - 1)

class ChiSquareDistribution(GammaDistribution):
    """
    Chi-square distribution is a special case of Gamma distribution where
    location = 0, scale = 2 and shape is twice that of the degrees of
    freedom.

    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4
    """

    def __init__(self, df=2):
        """
        Constructor method. The parameters are used to construct
        the probability distribution.
```

```
        @param df: degrees of freedom"""
        GammaDistribution.__init__(self, 0, 2, float(df) / 2.0)

class TDistribution(Distribution):
    """
    Class for Student's t-distribution.

    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4
    """

    def __init__(self, location=0.0, scale=1.0, shape=2):
        """Constructor method. The parameters are used to construct
        the probability distribution.

        @param location: default = 0.0
        @param scale: default = 1.0
        @param shape: degrees of freedom; default = 2"""
        self._mean = float(location)
        self.stdev = float(scale)
        self.df = float(shape)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity or 0
        to a give x-value on the x-axis where y-axis is the probability.
        """
        t = (x - self._mean) / self.stdev
        a = nrpy.betai(self.df / 2.0, 0.5, self.df / (self.df + (t * t)))
        if t > 0:
            return 1 - 0.5 * a
        else:
            return 0.5 * a

    def PDF(self, x):
        """
        Calculates the density (probability) at x with n-th degrees of
        Freedom as
        M{f(x) = S{Gamma}((n+1)/2) /
        (sqrt(n * pi) S{Gamma}(n/2)) (1 + x^2/n)^-((n+1)/2)}
        for all real x. It has mean 0 (for n > 1) and variance n/(n-2)
        (for n > 2)."""
        a = nrpy.gammln((self.df + 1) / 2)
        b = math.sqrt(math.pi * self.df) * nrpy.gammln(self.df / 2) * \
            self.stdev
        c = 1 + ((((x - self._mean) / self.stdev) ** 2) / self.df)
        return (a / b) * (c ** ((-1 - self.df) / 2))

    def inverseCDF(self, probability, start = -10.0,
                   end = 10.0, error = 10e-8):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis, together with
        the cumulative probability.
```

```
    @param probability: probability under the curve from -infinity
    @param start: lower boundary of calculation (default = -10)
    @param end: upper boundary of calculation (default = 10)
    @param error: error between the given and calculated probabilities
    (default = 10e-8)
    @return: Returns a tuple (start, cprob) where 'start' is the
    standard
    deviation for the area under the curve from -infinity to the given
    'probability' (+/- step). 'cprob' is the calculated area under the
    curve from -infinity to the returned 'start'.
    """

    # check for tolerance
    if abs(self.CDF(start)-probability) < error:
        return (start, self.CDF(start))
    # case 1: lower than -10 standard deviations
    if probability < self.CDF(start):
        return self.inverseCDF(probability, start-10, start, error)
    # case 2: between -10 to 10 standard deviations (bisection method)
    if probability > self.CDF(start) and \
    probability < self.CDF((start+end)/2):
        return self.inverseCDF(probability, start, (start+end)/2,
        error)
    if probability > self.CDF((start+end)/2) and \
    probability < self.CDF(end):
        return self.inverseCDF(probability, (start+end)/2, end, error)
    # case 3: higher than 10 standard deviations
    if probability > self.CDF(end):
        return self.inverseCDF(probability, end, end+10, error)
    # cprob = self.CDF(start)
    # if probability < cprob:
        # return (start, cprob)
    # while probability > cprob:
        # start = start + step
        # cprob = self.CDF(start)
    # return (start, cprob)


def mean(self):
    """Gives the arithmetic mean of the sample."""
    return self._mean

def mode(self):
    """Gives the mode of the sample."""
    return self._mean

def kurtosis(self):
    """Gives the kurtosis of the sample."""
    a = ((self.df - 2) ** 2) * nrpy.gammln((self.df / 2) - 2)
    return 3 * ((a / (4 * nrpy.gammln(self.df / 2))) - 1)

def skew(self):
    """Gives the skew of the sample."""
    return 0.0

def variance(self):
```

```
        """Gives the variance of the sample."""
        return (self.df / (self.df - 2)) * self.stdev * self.stdev

def test(statistic, distribution, confidence):
    """Generates the critical value from distribution and confidence value
    using the distribution's inverseCDF method and performs 1-tailed and
    2-tailed test by comparing the calculated statistic with the critical
    value.


    Returns a 5-element list
    [left result, left critical, statistic, right critical, right result]
    where
        - left result = True (statistic in lower critical region) or
        False (statistic not in lower critical region)
        - left critical = lower critical value generated from 1 -
        confidence
        - statistic = calculated statistic value
        - right critical = upper critical value generated from confidence
        - right result = True (statistic in upper critical region) or
        False (statistic not in upper critical region)

    Therefore, null hypothesis is accepted if left result and right result
    are both False in a 2-tailed test.

    @param statistic: calculated statistic (float)
    @param distribution: distribution to calculate critical value
    @type distribution: instance of a statistics distribution
    @param confidence: confidence level of a one-tail
        test (usually 0.95 or 0.99), use 0.975 or 0.995 for 2-tail test
    @type confidence: float of less than 1.0"""
    data = [None, None, statistic, None, None]
    data[1] = distribution.inverseCDF(1.0 - confidence)[0]
    if data[1] < statistic: data[0] = False
    else: data[0] = True
    data[3] = distribution.inverseCDF(confidence)[0]
    if statistic < data[3]: data[4] = False
    else: data[4] = True
    return data

def t1Mean(smean, pmean, svar, ssize, confidence):
    """
    Test 7: t-test for a population mean (population variance unknown)

    To investigate the significance of the difference between an assumed
    population mean and a sample mean when the population variance is
    unknown and cannot be assumed equal or not equal.

    Limitations
        1. Weaker form of Test 1

    @param smean: sample mean
    @param pmean: population mean
    @param svar: sample variance
    @param ssize: sample size
    @param confidence: confidence level
    """
```

```
    statistic = float((smean - pmean) / (svar / sqrt(ssize)))
    return test(statistic, TDistribution(shape = ssize-1), confidence)

def t2Mean2EqualVariance(smean1, smean2, svar1, svar2, ssize1, ssize2,
                         confidence, pmean1=0.0, pmean2=0.0):
    """
    Test 8: t-test for two population means (population variance unknown
    but equal)

    To investigate the significance of the difference between the means of
    two populations when the population variances are unknown but assumed
    equal.

    Limitations
        1. Weaker form of Test 2

    @param smean1: sample mean of sample #1
    @param smean2: sample mean of sample #2
    @param svar1: variances of sample #1
    @param svar2: variances of sample #2
    @param ssize1: sample size of sample #1
    @param ssize2: sample size of sample #2
    @param confidence: confidence level
    @param pmean1: population mean of population #1 (optional)
    @param pmean2: population mean of population #2 (optional)"""
    df = ssize1 + ssize2 - 2
    pvar = float((((ssize1 - 1) * svar1) + ((ssize2 - 1) * svar2)) / df)
    statistic = float(((smean1 - smean2) - (pmean1 - pmean2)) / \
                ((sqrt(pvar)) * sqrt((1.0 / ssize1) + (1.0 / ssize2))))
    return test(statistic, TDistribution(shape = df), confidence)

def t2Mean2UnequalVariance(smean1, smean2, svar1, svar2, ssize1, ssize2,
                           confidence, pmean1=0.0, pmean2=0.0):
    """
    Test 9: t-test for two population means (population variance unknown
    and unequal)

    To investigate the significance of the difference between the means of
    two populations when the population variances are unknown and unequal.

    Limitations
        1. Weaker form of Test 3

    @param smean1: sample mean of sample #1
    @param smean2: sample mean of sample #2
    @param svar1: variances of sample #1
    @param svar2: variances of sample #2
    @param ssize1: sample size of sample #1
    @param ssize2: sample size of sample #2
    @param confidence: confidence level
    @param pmean1: population mean of population #1 (optional)
    @param pmean2: population mean of population #2 (optional)"""
    statistic = float(((smean1 - smean2) - (pmean1 - pmean2)) / \
                sqrt((svar1 / ssize1) + (svar2 / ssize2)))
    df = float((((svar1 / ssize1) + (svar2 / ssize2)) ** 2) / \
        (((svar1 ** 2) / ((ssize1 ** 2) * (ssize1 - 1))) + \
            ((svar2 ** 2) / ((ssize2 ** 2) * (ssize2 - 1)))))
```

```
    return test(statistic, TDistribution(shape = df), confidence)

def tPaired(smean1, smean2, svar, ssize, confidence):
    """
    Test 10: t-test for two population means (method of paired
    comparisons)

    To investigate the significance of the difference between two
    population
    means when no assumption is made about the population variances.

    @param smean1: sample mean of sample #1
    @param smean2: sample mean of sample #2
    @param svar: variance of differences between pairs
    @param ssize: sample size
    @param confidence: confidence level"""
    statistic = float((smean1 - smean2) / (svar / sqrt(ssize)))
    return test(statistic, TDistribution(shape = ssize - 1), confidence)

def tRegressionCoefficient(variancex, varianceyx, b, ssize, confidence):
    """
    Test 11: t-test of a regression coefficient

    To investigate the significance of the regression coefficient.

    Limitations
        1. Homoedasticity of values

    @param variancex: variance of x
    @param varianceyx: variance of yx
    @param b: calculated Regression Coefficient
    @param ssize: sample size
    @param confidence: confidence level"""
    statistic = float(((b * sqrt(variancex)) / \
        sqrt(varianceyx)) * ((ssize-1) ** -0.5))
    return test(statistic, TDistribution(shape = ssize - 2), confidence)

def tPearsonCorrelation(r, ssize, confidence):
    """
    Test 12: t-test of a correlation coefficient

    To investigate whether the difference between the sample correlation
    coefficient and zero is statistically significant.

    Limitations
        1. Assumes population correlation coefficient to be zero (use Test
        13
        for testing other population correlation coefficient
        2. Assumes a linear relationship (regression line as Y = MX + C)
        3. Independence of x-values and y-values

    Use Test 59 when these conditions cannot be met

    @param r: calculated Pearson's product-moment correlation coefficient
    @param ssize: sample size
    @param confidence: confidence level"""
    statistic = float((r * sqrt(ssize - 2)) / sqrt(1 - (r **2)))
```

```
    return test(statistic, TDistribution(shape = ssize - 2), confidence)

def ChiSquarePopVar(values, ssize, pv, confidence = 0.95):
    """
    Test 15: Chi-square test for a population variance

    To investigate the difference between a sample variance and an assumed
    population variance.

    @param values: sample values
    @param ssize: sample size
    @param pv: population variance
    @param confidence: confidence level"""
    mean = sum(values)/ssize
    freq = [float((values[i] - mean) ** 2)
            for i in range(len(values))]
    sv = float((sum(freq)/(ssize-1)))
    statistic = float((((ssize - 1) * sv) / pv))
    return test(statistic, ChiSquareDistribution(df = ssize-1),
        confidence)

def FVarianceRatio(var1, var2, ssize1, ssize2, confidence):
    """
    Test 16: F-test for two population variances (variance ratio test)

    To investigate the significance of the difference between two
    population variances.

    @param var1: variance #1
    @param var2: variance #2
    @param ssize1: sample size #1
    @param ssize2: sample size #2
    @param confidence: confidence level"""
    statistic = float(var1 / var2)
    return test(statistic, FDistribution(df1 = ssize1 - 1,
        df2 = ssize2 - 1), confidence)

def F2CorrelatedObs(r, var1, var2, ssize1, ssize2, confidence):
    """
    Test 17: F-test for two population variances (with correlated
    observations)

    To investigate the difference between two population variances when
    there is correlation between the pairs of observations.

    @param r: Sample correlation value
    @param var1: variance #1
    @param var2: variance #2
    @param ssize1: sample size #1
    @param ssize2: sample size #2
    @param confidence: confidence level"""
    statistic = float(((var1 / var2)- 1) / (((((var1 / var2) + 1) ** 2) -
        \ (4 * (r ** 2) * (var1 / var2))) ** 0.5))
    return test(statistic, FDistribution(ssize1-1, ssize2-1), confidence)

def Chisq2Variance(ssize, svar, pvar, confidence):
    """
```

```
    Test 24: Chi-square test for an assumed population variance

    To investigate the significance of the difference between a population
    variance and an assumed variance value.

    Limitations
        1. Sample from normal distribution

    @param ssize: sample size
    @param svar: sample variance
    @param pvar: population variance (assumed)
    @param confidence: confidence level"""
    statistic = float((ssize-1) * svar/pvar)
    return test(statistic, ChiSquareDistribution(df = ssize - 1),
        confidence)

def F2Count(count1, count2, confidence, time1=0, time2=0, repeat=False):
    """
    Test 25: F-test for two counts (Poisson distribution)

    To investigate the significance of the difference between two counted
    results (based on a Poisson distribution).

    Limitations
        1. Counts must satisfy a Poisson distribution
        2. Samples obtained under same conditions.

    @param count1: count of first sample
    @param count2: count of second sample
    @param repeat: flag for repeated sampling (default = False)
    @param time1: time at which first sample is taken
        (only needed if repeat = True)
    @param time2: time at which second sample is taken
        (only needed if repeat = True)
    @param confidence: confidence level"""
    if not repeat:
        statistic = float(count1) / float(count2 + 1)
        numerator = 2 * (count2 + 1)
        denominator = 2 * count1
    else:
        statistic = (float(count1 + 0.5) / float(time1)) / \
                    (float(count2 + 0.5) / float(time2))
        numerator = 2 * count1 + 1
        denominator = 2 * count2 + 1
    return test(statistic, FDistribution(df1 = numerator,
        df2 = denominator), confidence)

def ChisqFit(observed, expected, confidence):
    """
    Test 37: Chi-square test for goodness of fit

    To investigate the significance of the differences between observed
    data arranged in K classes, and the theoretical expected frequencies
    in the K classes.

    Limitations
        1. Observed and theoretical distributions should have same number
```

```
        of elements
        2. Same class division for both distributions
        3. Expected frequency of each class should be at least 5

    @param observed: list of observed frequencies (index matched with
    expected)
    @param expected: list of expected frequencies (index matched with
    observed)
    @param confidence: confidence level"""
    freq = [float((observed[i] - expected[i]) ** 2) / (float(expected[i]))
            for i in range(len(observed))]
    statistic = 0.0
    for x in freq: statistic = statistic + x
    return test(statistic, ChiSquareDistribution(df = len(observed) - 1),
                confidence)

def tx2testofKcounts(T, V, confidence):
    """
    Test 38: The x2-test for compatibility of K counts

    To investigate the significance of the differences between K counts.

    Limitations:
        1. The counts must be obtained under comparable conditions

    @param T: list of time under K counts
    @param V: list of values of K counts
    @param confidence: confidence level"""
    R = float(sum(V)) / float(sum(T))
    freq = [((V[i] - (T[i] * R)) ** 2) / float(T[i] * R)
            for i in range(len(V))]
    statistic = sum(freq)
    return test(statistic, ChiSquareDistribution(df = len(V) - 1),
        confidence)

def Chisq2x2(s1, s2, ssize, confidence):
    """
    Test 40: Chi-square test for consistency in 2x2 table

    To investigate the significance of the differences between observed
    frequencies for two dichotomous distributions.

    Limitations
        1. Total sample size (sample 1 + sample 2) must be more than 20
        2. Each cell frequency more than 3

    @param s1: 2-element list or tuple of frequencies for sample #1
    @param s2: 2-element list or tuple of frequencies for sample #2
    @param ssize: sample size
    @param confidence: confidence level"""
    s1c1 = s1[0]
    s1c2 = s1[1]
    s2c1 = s2[0]
    s2c2 = s2[1]
    statistic = (float(ssize - 1) * ((float(s1c1 * s2c2) - \
        float(s1c2 * s2c1))**2)) / (float(s1c1 + s1c2) * float(s1c1 +
        s2c1) * \
```

```
        float(s1c2 + s2c2) * float(s2c1 + s2c2))
    return test(statistic, ChiSquareDistribution(df = 1), confidence)

def ChisquareKx2table(c1, c2, k, confidence):
    """Test 41: The x2-test for consistency in a K x 2 table

    To investigate the significance of the differences between K observed
    frequency distributions with a dichotomous classification.

    Limitations:
        1. K sample sizes must be large enough.
        2. It is usually assumed to be satisfied if the cell frequencies
        are equal to 5

    @param c1: class #1 values of sample k
    @param c2: class #2 values of sample k
    @param k: number of samples
    @param confidence: confidence level"""
    z = sum(c1)
    nx = sum(c2)
    n = [c1[i] + c2[i] for i in range(len(c1))]
    Total = sum(n)
    A = [x ** 2 for x in c1]
    B = sum([float(A[i]) / float(n[i]) for i in range(len(A))])
    C = (B - (float(z ** 2) / (float(Total))))
    D = float(Total ** 2) / float(z * (Total - z))
    statistic = D * C
    return test(statistic, ChiSquareDistribution(k-1), confidence)

def Chisquare2xKtable(s1, s2, k, confidence):
    """Test 43: The x2-test for consistency in a 2 x K table

    To investigate the significance of the differences between two
    distributions based on two samples spread over K classes

    Limitations:
        1. The two samples are sufficiently large
        2. The K classes when put together form a complete series

    @param s1: sample #1 of class k
    @param s2: sample #2 of class k
    @param k: number of classes
    @param confidence: confidence level"""
    N1 = sum(s1)
    N2 = sum(s2)
    Total = N1 + N2
    n =[s1[i] + s2[i] for i in range(len(s1))]
    e1 = [float(N1 * x) / float(N1 + N2) for x in n]
    e2 = [float(N2 * x) / float(N1 + N2) for x in n]
    A = [((s1[i] - e1[i]) **2) / e1[i] for i in range(len(s1))]
    B = [((s2[i] - e2[i]) **2) / e2[i] for i in range(len(s2))]
    statistic = sum([A[i] + B[i] for i in range(len(A))])
    return test(statistic, ChiSquareDistribution(k-1), confidence)

def MedianTestfor2Pop(s1, s2, confidence):
    """Test 50: The median test of two populations
```

```
    To test if two random samples could have come from two populations
    with same frequency distribution

    Limitations:
        1. The two samples are assumed to be reasonably large

    @param s1: 2-element list or tuple of frequencies for sample #1
    @param s2: 2-element list or tuple of frequencies for sample #2
    @param confidence: confidence level"""
    ls1 = s1[0]
    rs1 = s1[1]
    ls2 = s2[0]
    rs2 = s2[1]
    N = ls1 + ls2 + rs1 + rs2
    statistic = float(((abs((ls1*rs2)-(ls2*rs1))-(N * 0.5))**2)*N) / \
        ((ls1+ls2)*(ls1+rs1)*(ls2+rs2)*(rs1+rs2))
    return test(statistic, ChiSquareDistribution(df=1), confidence)
```

File: AllTestfile.py

```
import unittest
import AllTests as N

class testChiSquareDistribution(unittest.TestCase):

    def testChiSquarePopVar(self):
        """Test 15: Chi-square test for a population variance"""
        self.assertAlmostEqual(N.ChiSquarePopVar(values = (70, 71, 71, 69,
            69, 72, 72, 68, 68, 67, 67, 73, 73, 74, 74, 66, 66, 65, 65,
            75, 75, 76, 76, 64, 64), ssize = 25, pv = 9,
            confidence = 0.95)[2], 40.4, places = 1)
        self.assertTrue(N.ChiSquarePopVar(values = (70, 71, 71, 69, 69,
            72, 72, 68, 68, 67, 67, 73, 73, 74, 74, 66, 66, 65, 65, 75,
            75, 76, 76, 64, 64), ssize = 25, pv = 9,
            confidence = 0.95)[4])

    def testChisq2Variance(self):
        """Test 24: Chi-square test for an assumed population variance"""
        self.assertAlmostEqual(N.Chisq2Variance(ssize = 25, svar = 12,
            pvar = 9, confidence = 0.95)[2], 32)
        self.assertFalse(N.Chisq2Variance(ssize = 25, svar = 12,
            pvar = 9, confidence = 0.95)[4])

    def testChisqFit(self):
        """Test 37: Chi-square test for goodness of fit"""
        self.assertAlmostEqual(N.ChisqFit(observed = (25, 17, 15, 23, 24,
            16), expected = (20, 20, 20, 20, 20, 20), confidence =
            0.95)[2], 5.0)
        self.assertFalse(N.ChisqFit(observed = (25, 17, 15, 23, 24, 16),
            expected = (20, 20, 20, 20, 20, 20), confidence = 0.95)[4])

    def testtx2testofKcounts(self):
        """Test 38: The x2-test for compatibility of K counts"""
        self.assertAlmostEqual(N.tx2testofKcounts(T = (1, 1, 1, 1),
            V = (5, 12, 18, 19), confidence = 0.95)[2], 9.259, places = 2)
        self.assertTrue(N.tx2testofKcounts(T = (1, 1, 1, 1),
```

```
            V = (5, 12, 18, 19), confidence = 0.95)[4])

    def testChisq2x2(self):
        """Test 40: Chi-square test for consistency in 2x2 table"""
        self.assertAlmostEqual(N.Chisq2x2(s1 = (15, 85), s2 = (4, 77),
            ssize = 180, confidence = 0.95)[2], 4.79, places = 1)
        self.assertTrue(N.Chisq2x2(s1 = (15, 85), s2 = (4, 77), ssize =
            180, confidence = 0.95),[4])


    def testChisquareKx2table(self):
        """Test 41: The x2-test for consistency in a K x 2 table"""
        self.assertAlmostEqual(N.ChisquareKx2table(c1 = (3, 4, 5),
            c2 = (7, 2, 4), k = 3, confidence = 0.95)[2], 2.344, places =
            2)
        self.assertFalse(N.ChisquareKx2table(c1 = (3, 4, 5),
            c2 = (7, 2, 4), k = 3, confidence = 0.95)[4])

    def testChisquare2xKtable(self):
        """Test 43: The x2-test for consistency in a 2 x K table"""
        self.assertAlmostEqual(N.Chisquare2xKtable(s1 = (50, 47, 56),
            s2 = (5, 14, 8), k = 3, confidence = 0.95)[2], 4.84, places =
            2)
        self.assertFalse(N.Chisquare2xKtable(s1 = (50, 47, 56),
            s2 = (5, 14, 8), k = 3, confidence = 0.95)[4])

    def testMedianTestfor2Pop(self):
        """Test 50: The median test of two populations"""
        self.assertAlmostEqual(N.MedianTestfor2Pop(s1 = (9, 6),
            s2 = (6, 9), confidence = 0.95)[2], 0.53, places = 1)
        self.assertFalse(N.MedianTestfor2Pop(s1 = (9, 6), s2 = (6, 9),
            confidence = 0.95)[4])

class testFDistribution(unittest.TestCase):

    def testFVarianceRatio(self):
        """Test 16: F-test for two population variances (variance ratio
        test)"""
        self.assertAlmostEqual(N.FVarianceRatio(var1 = 0.36, var2 = 0.087,
            ssize1 = 6, ssize2 = 4, confidence = 0.95)[2], 4.14, places=2)
        self.assertFalse(N.FVarianceRatio(var1 = 0.36, var2 = 0.087,
            ssize1 = 6, ssize2 = 4, confidence = 0.95)[4])

    def testF2CorrelatedObs(self):
        """Test 17: F-test for two population variances
        with correlated observations)"""
        self.assertAlmostEqual(N.F2CorrelatedObs(var1 = 0.36,
            var2 = 0.087, ssize1 = 6, ssize2 = 6, r = 0.811,
            confidence = 0.95)[2], 0.796, places = 2)
        self.assertFalse(N.F2CorrelatedObs(var1 = 0.36, var2 = 0.087,
            ssize1 = 6, ssize2 = 6, r = 0.811, confidence = 0.95)[4])

    def testF2Count(self):
        """Test 25: F-test for two counts (Poisson distribution)"""
        self.assertAlmostEqual(N.F2Count(count1 = 13, count2 = 3,
            confidence = 0.95)[2], 3.25)
        self.assertTrue(N.F2Count(count1= 13, count2 = 3,
```

```
            confidence = 0.95)[4])

class testTDistribution(unittest.TestCase):

    def testt1Mean(self):
        """Test 7: t-test for a population mean (population variance
        unknown)"""
        self.assertAlmostEqual(N.t1Mean(smean = 3.1, pmean = 4.0, svar =
             1.0, ssize = 9, confidence = 0.975)[2], -2.69999999)
        self.assertFalse(N.t1Mean(smean = 3.1, pmean = 4.0, svar = 1.0,
            ssize = 9, confidence = 0.975)[4])

    def testt2Mean2EqualVariance(self):
        """Test 8: t-test for two population means (population variance
        unknown but equal)"""
        self.assertAlmostEqual(N.t2Mean2EqualVariance(smean1 = 31.75,
            smean2 = 28.67, svar1 = 112.25, svar2 = 66.64, ssize1 = 12,
            ssize2 = 12, confidence = 0.975)[2], 0.798, places=3)
        self.assertFalse(N.t2Mean2EqualVariance(smean1 = 31.75,
            smean2 = 28.67, svar1 = 112.25, svar2 = 66.64, ssize1 = 12,
            ssize2 = 12, confidence = 0.975)[4])

    def testt2Mean2UnequalVariance(self):
        """Test 9: t-test for two population means (population variance
        unknown and unequal)"""
        self.assertAlmostEqual(N.t2Mean2UnequalVariance(smean1 = 3166.0,
            smean2 = 2240.4, svar1 = 6328.67, svar2 = 221661.3, ssize1 =
            4, ssize2 = 9, confidence = 0.975)[2], 5.717, places=2)
        self.assertTrue(N.t2Mean2UnequalVariance(smean1 = 3166.0,
            smean2 = 2240.4, svar1 = 6328.67, svar2 = 221661.3, ssize1 =
            4, ssize2 = 9, confidence = 0.975)[4])

    def testtPaired(self):
        """Test 10: t-test for two population means (method of paired
        comparisons)"""
        self.assertAlmostEqual(N.tPaired(smean1 = 0.9, smean2 = 1.0, svar
            = 2.9, ssize = 10, confidence=0.975)[2], -0.11, places=2)
        self.assertFalse(N.tPaired(smean1 = 0.9, smean2 = 1.0, svar = 2.9,
            ssize = 10, confidence = 0.975)[4])

    def testtRegressionCoefficient(self):
        """Test 11: t-test of a regression coefficient"""
        self.assertAlmostEqual(N.tRegressionCoefficient(variancex = 15.61,
            varianceyx = 92.4, b = 5.029, ssize = 12, confidence =
            0.975)[2], 0.6232, places=3)
        self.assertFalse(N.tRegressionCoefficient(variancex = 15.61,
            varianceyx = 92.4, b = 5.029, ssize = 12, confidence =
            0.975)[4])

    def testtPearsonCorrelation(self):
        """Test 12: t-test of a correlation coefficient"""
        self.assertAlmostEqual(N.tPearsonCorrelation(r = 0.32, ssize = 18,
            confidence = 0.95)[2], 1.35, places=2)
        self.assertFalse(N.tPearsonCorrelation(r = 0.32, ssize = 18,
            confidence = 0.95)[4])
```

```
if __name__ == '__main__':
    unittest.main()
```

3. References

DOWNWARD, L., BOOTH, C. H., LUKENS, W. W. & BRIDGES, F. (2006) A Variation of the F-test for Determining Statistical Relevance of Particular Parameters in EXAFS Fits. *13th International Conference on X-ray Absorption Fine Structure***,** 129-131.

LING, M. (2009a) Ten Z-test Routines from Gopal Kanji's 100 Statistical Tests. The Python Papers Source Codes 1:5.

LING, M. (2009b) Compendium of Distributions, I: Beta, Binomial, Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform. The Python Papers Source Codes 1:4.

MCDONALD, J. H. (2008) *Handbook of Biological Statistics* Baltimore, Sparky House Publishing.

SIRKIN, R. M. (2006) *Statistics for the Social Sciences*, Sage Publications Inc.